

A New Algorithm for Interactive Graphics on Multicomputers

David A. Ellsworth
University of North Carolina at Chapel Hill

This polygon rendering algorithm uses a new load-balancing method that exploits the frame-to-frame coherence of interactive applications.

As nonshared-memory multiple instruction, multiple data (MIMD) systems become more common, it becomes important to develop parallel rendering algorithms for them. These systems, known as multicomputers, can produce data sets so large that it is difficult to visualize the data on conventional graphics systems, especially if the visualization proceeds in tandem with the calculation. Parallel systems must run interactive graphics to allow convenient visualizations of their computations. While few parallel systems currently have a frame buffer that will support interactive rendering, such systems should be more common in the future.

This article describes an algorithm suited for interactive polygon rendering, where the model's image on screen generally has frame-to-frame coherence. The algorithm uses this coherence to perform load-balancing calculations in parallel with the other calculations. The algorithm also uses an optimized version of personalized all-to-all communication, where all processors communicate with all other processors.

The following sections briefly talk about previous work and the trade-offs involved in designing a parallel rendering algorithm. Later sections describe the new algorithm in these contexts, an implementation of it, and performance results.

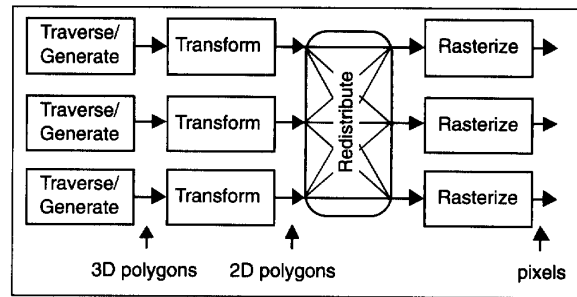
Related work

Many people have investigated ways to parallelize the rendering process. One method is to split the screen into contiguous regions and statically assign the regions to processors. Parke,¹ Kaplan and Greenberg,² Whelan,³ and Whitman⁴ all studied variations of this method. Whitman also studied methods of statically dividing the screen into regions, then assigning the regions to processors as the frame progresses, automatically load balancing the rendering work.⁵ This method was implemented in Pixel-Planes 5, a hardware system.⁶

Whelan, Whitman, and Roble⁷ studied adaptively dividing the screen according to the distribution of polygons on the screen, balancing the processing needed in each portion. Ortega et al.⁸ recently implemented a data-parallel algorithm that achieves high performance and interactive rates.

The starting point for the algorithm presented in this article was the work of Crockett and Orloff.⁹ They described an algorithm implemented on an Intel iPSC/860 that statically assigns contiguous groups of scan lines to processors. They also investigated methods to overlap the transformation and rendering tasks, and developed a performance model for their algorithm.

Figure 1. A fully parallelized graphics pipeline.



Parallel rendering algorithms

This section describes the variations in rendering algorithms considered during the design of the new algorithm presented in the next section. To establish a common ground, let us first discuss how the graphics pipeline maps onto a general parallel algorithm. Consider a graphics pipeline consisting of the following tasks:

1. Traverse a stored graphics database (retained mode) or generate polygons (immediate mode).
2. Transform the polygons to screen space, clip them, and light the vertices.
3. Rasterize the polygons: scan convert, remove hidden surfaces, and interpolate the shading.

Figure 1 shows a parallelized version of the graphics pipeline. Like the standard graphics pipeline, the parallel pipeline specifies the flow of data and the operations to perform on them, but not when to perform the operations. The first two tasks of the parallel pipeline generate and transform polygons. Together, they constitute the *geometry processing* stage. If the application uses a retained database, the database can be broken up and stored in the processors' memories.¹⁰ In the simplest implementation, this puts the first polygon on the first processor, the second polygon on the second processor, and so forth, so each processor has nearly the same number of polygons. State changes, like color changes and transformation matrix changes, are replicated on each processor. With some additional work,¹⁰ the superfluous state changes can be removed.

On the other hand, if the application uses an immediate mode interface, the graphics library cannot partition and load-balance the geometry processing stage. Instead, the application does the partitioning, because it controls the generation process. The load balancing of the transformation follows from the application's load balancing of polygon generation. As the polygons are traversed or generated, they are transformed into screen space and shaded according to the selected lighting model.

The third task, rasterization, subdivides the screen into regions, then assigns the regions to processors so that each has the same amount of work to do. The redistribution stage (between the geometry processing and rasterization stages) reorganizes the polygons between the different partitions used in the two stages. The redistribution must be a global sort, as any polygon can affect any part of the screen.

The transformation task classifies each polygon according to the regions it overlaps and sends it to the processors assigned to those regions. For example, the triangle in Figure 2 would be sent to the processors assigned to the four shaded regions. The average number of times each triangle is sent to a processor is called the *overlap factor*. The overlap factor is generally small for small polygons. If the screen is divided into 256 regions, the models used later in this article would have overlap factors ranging from 1.09 to 1.4. Another article in this issue (Molnar et al.,¹¹

pp. 23-32) gives more details about the overlap factor, including a formula for calculating the expected overlap factor.

Parallelizing the rasterization

So far, I have described methods for parallelizing the geometry processing tasks. Here I describe three aspects of parallelizing the rasterization.

Subdividing the screen

There are many ways to subdivide the screen. Two parameters are the shape and the number of regions. Square regions have been shown to give the smallest overlap factor for randomly oriented polygons.^{3,4} Intuitively, we can see that a long, thin region would produce a higher overlap factor than a square region with the same area.

Finding the best number of regions involves a trade-off. On one hand, increasing the number of regions improves the load balance of the rasterization. If only $1/n$ of the screen has polygons covering it, then we need at least n regions per processor so that each processor has some work to do. To have a good load balance, we need more regions per processor. On the other hand, increasing the number of regions increases the number of times each polygon must be communicated. It also increases the time spent on per-polygon rasterization overhead, that is, the preparation work involved for rasterizing a polygon. We must balance these two factors to find the best number of regions per processor, or *granularity ratio*.

Scheduling the algorithm stages

We can schedule three stages of the pipeline: the geometry processing, the redistribution, and the rasterization. Clearly, the redistribution should overlap the other computations as much as practicable. Otherwise, the parallelism in having both resources working would be lost. Two current multicomputer rendering algorithms use overlapped communication.^{6,9}

Geometry processing and rasterization can be run concurrently or consecutively. Crockett and Orloff's implementation⁹ overlaps them throughout the frame—the active stage switches every few polygons. To reduce communication costs, an algorithm must buffer triangles by sending messages that contain more than one triangle. Buffering amortizes the per-message cost over several triangles. However, it can also cause congestion at the end of the frame when the partially filled polygon message buffers are flushed. The congestion is worst when the load-balancing algorithm is most successful—that is, when all the processors finish the frame at the same time.

The second possibility is to run stages consecutively—first the geometry processing and then the rasterization. This moves the network congestion caused by buffer flushing from the end

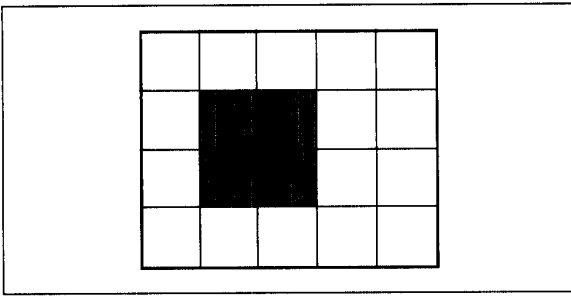


Figure 2. Polygon redistribution example. The triangle will be sent to the processors assigned to the four shaded regions.

of the frame to the middle, allowing time for the congestion to clear before the processors need to wait for late-arriving polygons. This approach comes at the cost of the memory to store the transformed polygons before rasterization.

Load balancing

The load-balancing algorithm and the time to run it depend on the scheduling of the geometry processing and rasterization stages. I describe three scheduling options below. If the two stages run consecutively, all three options are possible. If they run concurrently, only the last option is possible. The options are

1. *Between rendered regions, using a work-queue approach.* When a processor starts rendering, it is assigned a region and renders it. When it finishes the region, it asks for another region and continues this loop until the frame is done. However, each assignment must be sent to all processors so that each processor can send its portion of the polygons for that region to the designated processor. This increases the minimum number of messages required, so it is expensive on systems with large per-message costs. The communication is not necessary on shared-memory systems, as a processor can retrieve data with minimal cost to the other processor. Both Whitman⁴ and Pixel-Planes 5⁶ use this method.
2. *Between the two stages.* After all the processors finish transforming, the number of polygons in each region is collected from all processors and the regions are assigned to processors on the basis of this count. The assignments are broadcast, then the processors send their screen-space polygons and start rendering. This introduces a synchronization point and some serial computation in the middle of the frame, limiting the amount of parallelism possible. Roble⁷ used this approach.
3. *Between frames.* Like the previous option, this one collects per-region polygon counts after the processors finish transforming. However, this option uses the assignment based on this count for the *next* frame. Thus, this method depends on frame-to-frame coherence in the distribution of polygons over the screen. This assumption is reasonable for interactive applications, because not having such coherence would make it hard for people to follow the output images. This method allows the processors to start rasterizing immediately after they finish transforming. Also, the processors can use the entire frame to send the polygons because the polygon destinations have already been determined. This assumes a z-buffer algorithm; a scan-line algorithm must wait for all the polygons to arrive before starting rasterization.

These three load-balancing methods are not exclusive, as hybrid methods are possible. For example, you could use the third method to assign some regions and the first method to assign the remainder.

One advantage of using the third method with consecutive transformation and rasterization is that it moves the load balancing out of the critical path. Once the transformation is finished, the next frame's region assignment is computed in parallel with the rasterization. It is usually ready by the time the rasterization is completed.

Simulation

I used a simulator to help determine a granularity ratio and when to do the load balancing. The simulator calculates the expected processor utilization during rasterization. It assumes no time is lost waiting for data. During a run, the simulator reads in a static model and renders it from a series of recorded viewpoints. It transforms each frame, classifies polygons by region, computes the region-to-processor assignments, then calculates the processor utilization expected for the frame's rasterization stage.

The assignment algorithm determines the per-region costs and then uses a greedy multiple-bin-packing algorithm to make the assignments. First, the polygon counts are sorted. Then, the regions are assigned, starting with the one having the largest polygon count and proceeding to the smallest. Each region is assigned to the processor that currently has the lightest load. The smallest regions are assigned last so they "fill in" any unevenness created when assigning the heavily loaded regions.

The simulated region-assignment algorithm uses an estimate of the time that each region will take to render so that it can balance the rasterization loads. A region's estimated rasterization time is the sum of the region's overhead time (the time to process a region with no triangles) and a simple estimate of the time to render each polygon overlapping the region, based only on the number of vertices in the polygon. On the other hand, when calculating processor utilization, the simulator uses a more accurate time estimate based on the number of pixels in each triangle and the number of scan lines crossed. I derived the costs and times from performance runs.

I performed the simulation on two models, each with a series of recorded viewing positions. The first model is the National Computer Graphics Association (NCGA) Graphics Performance Committee's Program Level Benchmark (PLB) head model and viewing positions. The benchmark's viewing positions have the head rotate 4.5 degrees to the left for each frame, with a total of 80 frames. The second model is the outer shell of the polio virus. The series of 110 viewing positions were recorded as the model was interactively displayed on Pixel-

Figure 3. Representative views of the PLB head model (left) and the polio model (right). The two models have 59,592 triangles with an average size of 4.4 pixels and 806,640 triangles with an average size of 6.7 pixels, respectively.

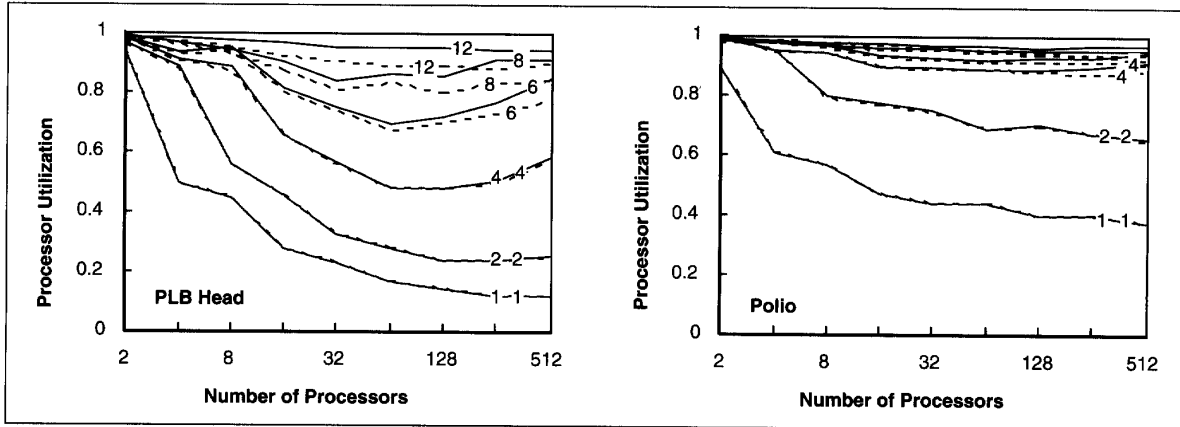
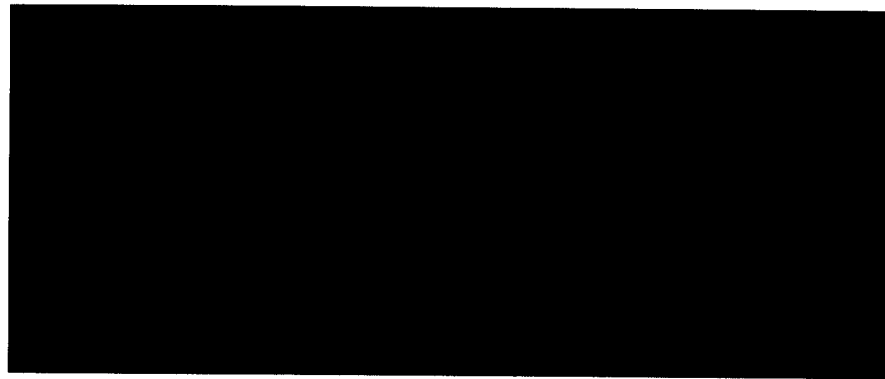


Figure 4. Results from simulating processor utilization for the rasterization stage. Each curve gives the utilization for the indicated granularity ratio. For better clarity, the chart for the polio model omits labels for the top three pairs of curves. These curves should be labeled 6, 8, and 12, numbered from bottom to top.

Planes 5. Figure 3 shows representative views of the models.

A series of simulation runs for each model sampled three independent parameters: the number of processors (2 to 512), the granularity ratio (1 to 12), and the time when polygon counts are sampled (during the current or previous frame). The simulation culled backfacing polygons.

Figure 4 presents the results. Each curve shows the expected processor utilization for a constant granularity ratio across a range of system sizes. The curves in red show the utilization when using the current frame's per-region polygon counts, and the curves in black show the results using the previous frame's information. As expected, the utilization increases as the granularity ratio increases. The figure also shows a decrease in utilization as the number of processors increases. This is due to the increase in variance in the region polygon counts as the number of regions increases. Compared to smaller numbers of processors, large numbers of processors need a larger granularity ratio to average out the increased variance. As illustrated by the differences between the two graphs, the amount of region-to-region variance is scene-dependent.

Finally, the simulation results show a fairly small penalty for using the previous frame's polygon counts during the region assignment. For 512 processors, the drop in utilization ranges only up to 7 percent. A simulation for a different model (mentioned below "Performance") predicted a drop of 12 percent.

Estimated total time

The utilization graphs give only part of the information needed to choose the granularity ratio. The processing time associated with the overlap must be factored in. I did this by using a performance model that estimates the rendering time. The model employs utilization and overlap factor values from the simulator plus measured calculation and communication times from the Touchstone Delta. The model does not include time for network congestion or performing the load balancing.

Figure 5 presents the results of the performance model for each of the two models. Each graph plots the relative execution time for a range of granularity ratios (2 to 12) and a range of processor numbers (2 to 512). The graphs plot the relative execution time instead of the actual execution time so that normal speedup as processors are added does not obscure the result. I computed the relative execution times by dividing each time by the average execution time of all cases with the same model and number of processors.

The figure shows that a granularity ratio of four would work best for the polio model, and a ratio of 12 would work best for the head model. It is not surprising that different models would work best with different ratios. The expected optimum number depends on the distribution of polygons on the screen: Models with heavier concentrations need a larger granularity ratio so that the polygon concentrations can be broken into tasks that

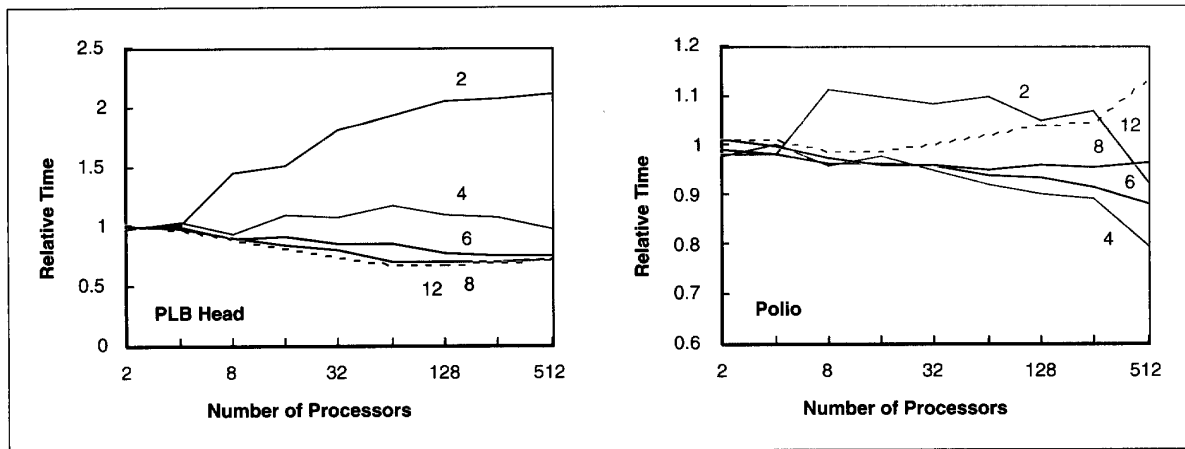


Figure 5. Predicted normalized rendering times for the PLB head and polio models.

```

if (processor 0) {
    compute region assignments
    broadcast region assignments
}
else
    read region assignments
synchronize /* ensure prev. frame finished */
for (each polygon) {
    transform to screen space
    place in buffer(s) for processor(s)
    send buffer if full
}
flush polygon buffers
initialize color- and z-buffer for my regions
while (there are more polygons to process) {
    get a buffer full of polygons
    render each polygon into appropriate
    frame buffer
}
send pixels to frame buffer and/or disk

```

a single processor can handle. The best ratio will also vary depending on the system used. For example, a system with high communication cost would favor a smaller granularity ratio, trading communication for processor utilization.

Algorithm summary

Given the previous discussion and simulation results, I chose the following algorithm features:

- The transformation and rasterization stages are executed sequentially instead of being overlapped.
- Load balancing figures are derived from the previous frame's per-region polygon counts.
- The granularity ratio is fixed at eight regions per processor—a compromise between the optimum values for the two models.

To fill in the details, Figure 6 presents a pseudocode version of the algorithm. The algorithm does not show incoming poly-

Figure 6. Parallel rendering algorithm pseudocode.

gons because they are handled asynchronously by a message handler (an interrupt-driven receive routine). When a polygon message arrives, the handler saves it for later processing.

The algorithm performs the load-balancing calculations concurrently with the rasterization step. When a processor flushes its buffers, it also enables an asynchronous handler that participates in a summing tree. This tree sums the per-region polygon counts and sends them to processor 0. Processor 0 then assigns regions to processors using the same multiple-bin-packing algorithm employed by the simulator. The first frame's load balancing is done by assigning each region r to the processor numbered $r \bmod N$.

The load-balancing algorithm takes into account the time spent by processor 0 calculating the region assignments. It also adds in the time spent on overhead for each region (that is, clearing the color buffer and z-buffer, and sending the computed pixels to the frame buffer and disk). It uses different times when different algorithm options are selected (for example, antialiasing and two-step redistribution). Having the load-balancing algorithm operate on expected times allows it to load-balance unrelated tasks (polygon rasterization and the load balancing itself) in the same operation. This gives flexibility to the implementation, but at the cost of having to measure the expected times whenever the implementation is modified or ported to a different system.

The algorithm described here is similar to the one I presented at the 1993 Parallel Rendering Symposium. The main difference is that each processor now sends a minimum of one message to each other processor instead of one message for each region. The old algorithm only had polygons from one region in each message, while the new algorithm will mix polygons for any region that the destination processor rasterizes. The new method reduces the minimum number of messages each processor must send for each frame. However, it increases the memory requirements when using small numbers of processors, since each processor allocates a color and z-buffer for all the regions it rasterizes instead of sharing one set of buffers for its regions.

Figure 7. Sixteen processors placed in four groups for two-step redistribution. The thick line separates groups.

Two-step redistribution

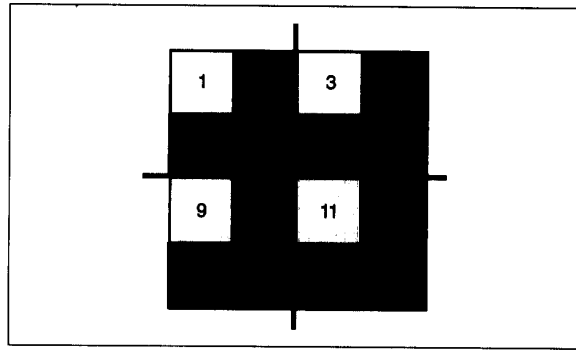
A separate feature of this algorithm optimizes the redistribution step, which the parallel algorithm community has studied as personalized all-to-all communication.¹² The optimization is needed because all-to-all communication does not scale very well. As the size of the system grows, the minimum number of messages grows even more quickly. A growth in system size by a factor of n increases the minimum number of messages by a factor of n^2 . This means that the average message size shrinks as the number of processors increases—even when the model size is scaled with the number of processors. Eventually, the per-message overhead dominates the communication cost.

The solution to this problem is to send the transformed polygons hierarchically. The polygons are sent twice, instead of once, so they take two steps before they reach the final processor. In the first step, the polygons are sent most of the way, to a forwarding processor. In the second step, they are sent to the final destination. The method requires the processors to be divided into groups. For processors organized into a 2D mesh, this grouping splits the processors into regular rectangular groups, where all the cuts are made horizontally or vertically across the entire array of processors. Figure 7 shows an example of 16 processors split into four groups.

Each processor is assigned one forwarding processor per group. In the figure, a processor's forwarding processors are the ones with the same color. In the first step, each processor sends only to its forwarding processors. When a processor receives polygons from the first step, it copies them into buffers for the second step. A processor maintains one buffer for each of the processors in its group, sending the buffer to that processor when it is filled. (Because the forwarding uses asynchronous message handlers, it does not appear in the pseudocode in Figure 6.)

While each polygon is sent twice, the bisection bandwidth only increases slightly, if at all. In the example, the bisection point happens to fall on a group cut point, so that bisection bandwidth does not change. When the bisection falls in the middle of a group (the usual case), the overall bisection bandwidth increases by the much smaller group bandwidth. The smaller number of messages used in the two-step redistribution works to increase the available bisection bandwidth because less time is spent on message overhead.

How many groups should be used? If you allow messages of arbitrary length, it is simple to compute that the number of messages sent is $N(g-1) + N^2/g - N$, where N is the number of processors and g is the number of groups. In the first step, each processor sends a message to all the groups except its own, giving $g-1$ messages per processor. In the second step, each processor sends a message to the N/g processors in the group, except that no processor sends a message to itself. The minimum number of messages occurs when $g = \sqrt{N}$. When the number of groups does not divide the number of processors evenly, the algorithm tries to assign groups and forwarding processors so that each processor sends and receives roughly the same number of messages.



The two-step redistribution method must work with the load-balancing algorithm. When processor 0 spends all of the rendering time calculating the processor region assignments, it is not assigned any regions to render. When this happens, processor 0 is also relieved from forwarding messages.

In addition to reducing the number of messages, two-step redistribution reduces the number of polygon buffers required, which reduces the memory required by the buffers if fixed-size buffers are used. Instead of requiring a buffer for every other processor, a processor needs buffers only for each of the other groups and for each of its own group members.

Two-step redistribution should be used only in cases where it saves time. An implementation can determine this automatically by evaluating a performance model.

Algorithm implementation

I implemented this algorithm on the Touchstone Delta at the California Institute of Technology. This prototype multicomputer has 512 Intel i860 compute nodes arranged in a 16×32 2D mesh. The compute node mesh is embedded in the center of a 16×36 mesh, with 64 I/O nodes surrounding the computing nodes. The links between the nodes are bidirectional and support up to 17 Mbytes per second in each direction.

The implementation uses the Gouraud shading model, with 24 bits of color, plus a 32-bit z-buffer. The inner loops—triangle transformation and rasterization—are written in assembly language using the Intel i860 dual-operation and dual-instruction modes and the graphics instructions. The implementation also supports antialiasing using 16 samples per pixel.

The buffer size was fixed at 4 Kbytes. The size was chosen to avoid excessive memory usage and to keep the per-message overhead reasonable. With 4-Kbyte buffers, the overhead is one-third of the overall cost. The buffers are sent asynchronously to reduce problems with message congestion.

Performance

I measured performance of the implementation as it displayed a sequence of views of the two models used in the simulation described earlier. I also measured performance for two other models and viewing sequences, shown in Figure 8. One model is of terrain generated from Landsat data. The other model is of a child's head, generated from computerized tomographic (CT) volume data by a marching cubes program.

To measure the algorithm's performance, I chose not to save the computed pixels, because sending them to disk—or even to the frame buffer—would have constrained the maximum frame rate (the Delta frame buffer is limited to about four updates per

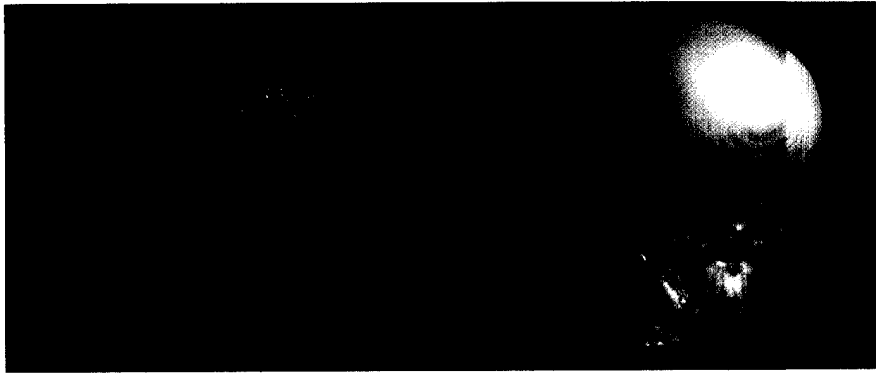


Figure 8. The terrain model (left) and the CT head model (right). The terrain model has 162,690 triangles with an average size of 2.8 pixels; the CT head model has 229,208 triangles with an average of 4 pixels.

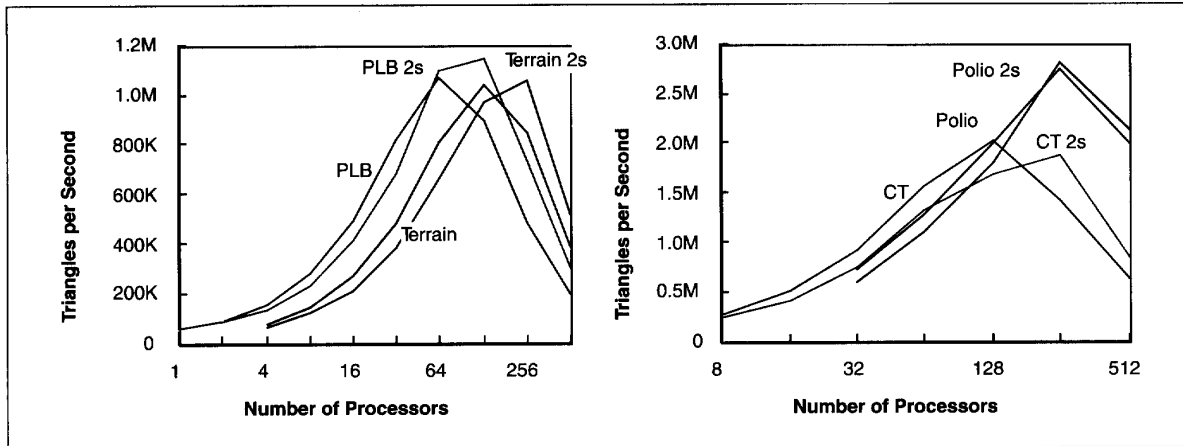


Figure 9. Rendering performance for the four models. PLB indicates the PLB head model, and CT indicates the CT head model ("2s" indicates use of two-step redistribution).

second). I computed the images at a resolution of 640×512 , with antialiasing disabled. The triangles-per-second values were computed by dividing the number of triangles in the viewing frustum by the frame time. Then I averaged the values across all the frames in the sequence.

I measured the performance for four models running on a variety of processor partition sizes. Memory limitations prevented runs with the large data sets when using a small number of processors. Figure 9 shows the performance measured with two-step redistribution both enabled and disabled. Note that the terrain model shows a lower performance value because it was rendered with backface culling disabled, while the other three were rendered with backface culling enabled.

Discussion

The performance graphs show three main results. First, the algorithm gives high performance, in some cases faster than most commercial workstations, especially with 128 and 256 processors. Second, the two-step redistribution is an improvement in cases with small data sets and many processors. The largest performance increase is 50 percent.

The third result is that performance reaches a peak, then drops off as the number of processors increases. This arises from two factors. The first affects the large models: The communication network saturates when running at high triangles-per-second rates. When running at 2.8 million triangles per

second on a 256-node partition, each link that crosses the partition bisection is carrying an average 2.5 Mbytes/s, which is a quarter of the maximum rate for 4-Kbyte messages. The system can only achieve the maximum rate when there is no contention for the links and when the link is used continuously, neither of which is true. One indication of the congestion is that processors near the center of the partition spend much longer waiting for polygons to arrive than ones near the edges.

The main cause of the peaks in performance is that the load-balancing algorithm limits the maximum frame rate. The main delay occurs during the collection of per-region polygon counts from all the processors: When using a large number of processors, the larger messages simply do not traverse the collection tree fast enough. The network contention exacerbates the delay.

Figure 10 shows speedup factors, with the performance values in Figure 9 divided by the performance of a single processor rendering the PLB head. The speedup values correspond to low parallel efficiencies, which have several causes. Two causes are the network contention and frame-rate limitation, already discussed. Third is the optimized assembly language code for the serial part of the implementation. The assembly code magnifies the overhead of the parallel algorithm, as the time spent in communication is a higher percentage than it would be otherwise. With 128 processors and the polio data set, approximately 40 percent of the CPU time is used for communication.

The antialiasing performance is somewhat surprising. With

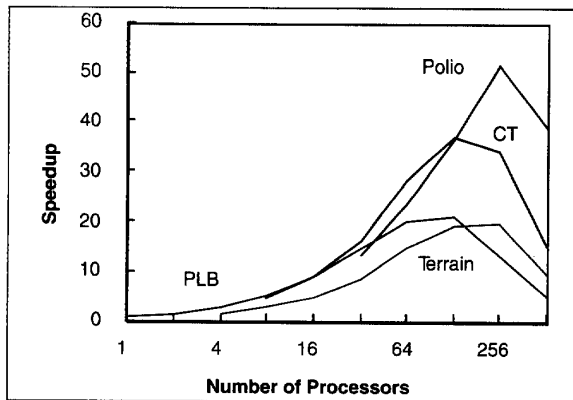


Figure 10. Speedup values. The speedup shown is the larger speedup of two cases, namely, with and without two-step redistribution.

small partitions (eight processors), the antialiased triangles-per-second performance is 36 percent of the aliased performance. However, the parallel bottlenecks with larger partitions decrease the net cost, since time that would otherwise be spent waiting is instead put to use. With 512 processors, and with 256 processors for all models except polio, the antialiased performance equals the aliased performance.

Conclusions and future work

The new algorithm uses frame-to-frame coherence to perform the load balancing in parallel with other computations. It does not scale well due to the load-balancing method and network congestion. The higher capacity networks of newer multi-computers such as the Intel Paragon should reduce these problems. Alternatively, the load-balancing bottleneck for large numbers of processors on the Delta could be removed by a static assignment of regions to processors. The loads would not be as balanced, but the performance would probably be higher than the current results for 256 and 512 processors.

Future work includes using the adaptively sized buffers developed by Crockett and Orloff,⁹ incorporating more accurate estimates of the time to render a polygon instead of assuming a constant cost, and investigating adaptive screen space methods.

Still, the algorithm achieves higher performance than any current multicomputer implementation I know, with a peak of 2.8 million triangles per second point-sampled and 2.1 million triangles per second antialiased. The Delta implementation achieved both peaks using 256 processors with the polio model. The implementation also achieved interactive frame rates, with a peak speed of 19 frames per second for the PLB head model with 128 processors. □

Acknowledgments

This work was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to the Delta was provided by Steve Taylor and Paul Messina of Caltech and by the Advanced Research Projects Agency. Thanks go to Carl Mueller for his help in preparing this article, to the anonymous reviewers for their comments, and to Tom Crockett for suggesting the new communication strategy.

The PLB head model appears courtesy of the IBM AWD Graphics Lab, Austin, Texas; the terrain model courtesy of H. Towles, Sun Microsystems; and the polio model courtesy of James Hogle, Marie Chow, and David Filman, Research Institute of Scripps Clinic. Franz Zonneveld, Utrecht University Hospital, The Netherlands, provided the CT

head volume data.

This work was sponsored by ARPA Information Systems Technology Office order 7510, National Science Foundation grant MIP-9000894, and the Science and Technology Center for Computer Graphics and Scientific Visualization (NSF cooperative agreement ASC-8920219).

References

1. F.I. Parke, "Simulation and Expected Performance Analysis of Multiple Processor Z-buffer Systems," *Computer Graphics (Proc. Siggraph)*, Vol. 14, No. 3, July 1980, pp. 48-56.
2. M. Kaplan and D.P. Greenberg, "Parallel Processing Techniques for Hidden Surface Removal," *Computer Graphics (Proc. Siggraph)*, Vol. 13, No. 2, Aug. 1979, pp. 300-307.
3. D.S. Whelan, *Animac: A Multiprocessor Architecture for Real-Time Computer Animation*, doctoral dissertation, California Institute of Technology, Pasadena, Calif., 1985.
4. S. Whitman, *Multiprocessor Methods for Computer Graphics Rendering*, AK Peters, Wellesley, Mass., 1992.
5. S. Whitman, "A Task-Adaptive Parallel Graphics Renderer," *IEEE CG&A*, Vol. 14, No. 4, July 1994, pp. 41-48.
6. H. Fuchs et al., "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Computer Graphics (Proc. Siggraph)*, Vol. 23, No. 3, July 1989, pp. 79-88.
7. D.R. Roble, "A Load Balanced Parallel Scanline Z-buffer Algorithm for the iPSC Hypercube," *Proc. First Int'l Conf. Pixim 88*, Editions Hermes, Paris, 1988, pp. 177-192.
8. F.A. Ortega, C.D. Hansen, and J.P. Ahrens, "Fast Data Parallel Polygon Rendering," *Proc. Supercomputing 93*, IEEE Computer Society Press, Los Alamitos, Calif., 1993, pp. 709-718.
9. T. W. Crockett and T. Orloff, "A MIMD Rendering Algorithm for Distributed Memory Architectures," *Proc. Visualization 93 Symp. Parallel Rendering*, ACM Press, New York, 1993, pp. 35-42.
10. D.A. Ellsworth, H. Good, and B.W. Tebbs, "Distributing Display Lists on a Multicomputer," *Computer Graphics (Proc. Symp. Interactive 3D Graphics)*, Vol. 24, No. 2, Mar. 1990, pp. 147-155.
11. S.E. Molnar et al., "A Sorting Classification of Parallel Rendering," *IEEE CG&A*, Vol. 14, No. 4, July 1994, pp. 23-32.
12. V. Kumar et al., *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms*, Benjamin Cummings, Redwood City, Calif., 1994.



David A. Ellsworth is a doctoral candidate in the Computer Science Department at the University of North Carolina at Chapel Hill. His research interests include parallel rendering, performance modeling, and parallel algorithms. Ellsworth received a BS in electrical engineering and computer science from the University of California at Berkeley and an MS in computer science from the University of North Carolina at Chapel Hill.

Readers can contact Ellsworth at the University of North Carolina, Computer Science Dept., CB 3175 Sitterson Hall, Chapel Hill, NC 27599, e-mail ellsworth@cs.unc.edu.